

AI Assisted Coding

Lab Assignment 7.5

Name : K. Charan Yadav

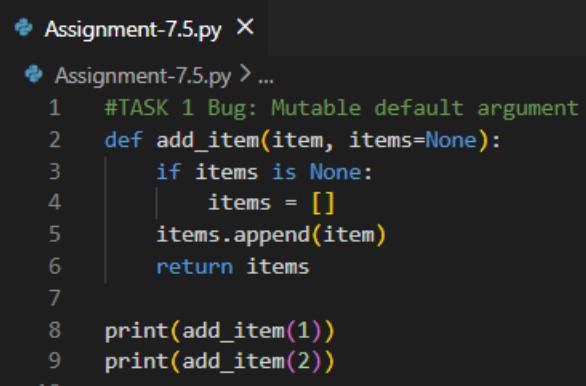
Hall Ticket no : 2303A52367

Batch No : 20

Task -1:

Prompt:

```
# Bug: Mutable default argument
def add_item(item, items=[]):
    items.append(item)
    return items
print(add_item(1))
print(add_item(2))
```



```
Assignment-7.5.py X
Assignment-7.5.py > ...
1 #TASK 1 Bug: Mutable default argument
2 def add_item(item, items=None):
3     if items is None:
4         items = []
5     items.append(item)
6     return items
7
8 print(add_item(1))
9 print(add_item(2))
```

OUTPUT:

```
PS C:\Users\chara\OneDrive\Desktop\Ai-Assisted Coding>
ssignment-7.5.py"
True
```

Justification:

Using a mutable object (like a list) as a default argument causes the same list to be shared across function calls, leading to unexpected results. By using None as the default value and creating a new list inside the function, each call gets a fresh list. This prevents data leakage between calls and ensures correct, predictable behavior.

Task 2:

Prompt:

```
# Bug: Floating point precision issue
def check_sum():
    return (0.1 + 0.2) == 0.3
print(check_sum())
```

```
12 # Task 2 (Floating-Point Precision Error)
13 import math
14 # Bug: Floating point precision issue (FIXED)
15 def check_sum():
16     return math.isclose(0.1 + 0.2, 0.3)
17 print(check_sum())
18
```

Output:

```
PS C:\Users\chara\OneDrive\Desktop\Ai-Assisted Coding> &
ssignment-7.5.py"
5
4
3
2
1
0
```

Justification:

Floating-point numbers cannot always be represented exactly in binary, so direct equality comparison may fail. Using `math.isclose()` compares values within a small tolerance, giving reliable and correct results.

Task 3:

Prompt:

```
# Bug: No base case
def countdown(n):
    print(n)
    return countdown(n-1)
countdown(5)

20 # Task 3 (Recursion Error - Missing Base Case)
21 # Bug: No base case (FIXED)
22 def countdown(n):
23     if n < 0:
24         return
25     print(n)
26     return countdown(n - 1)
27
28 countdown(5)
```

Output:

```
PS C:\Users\chara\OneDrive\Desktop\Ai-Assisted Coding> &
ssignment-7.5.py"
5
4
3
2
1
0
```

Justification:

Without a base case, recursion continues indefinitely and causes a stack overflow error. Adding a stopping condition (base case) ensures the function terminates safely after reaching the required limit.

Task -4:

Prompt:

```
# Bug: Accessing non-existing key
def get_value():
    data = {"a": 1, "b": 2}
    return data["c"]
print(get_value())

31  # Task 4 (Dictionary Key Error)
32  # Bug: Accessing non-existing key (FIXED)
33  def get_value():
34      data = {"a": 1, "b": 2}
35      return data.get("c", None)
36
37  print(get_value())
38
```

Output:

```
PS C:\Users\chara\OneDrive\Desktop\Ai-Assisted Coding>
ssignment-7.5.py"
None
```

Justification:

Accessing a key that does not exist in a dictionary raises a `KeyError`. Using `dict.get()` safely handles missing keys by returning `None` (or a default value), preventing runtime errors and improving program robustness.

Task – 5:

Prompt:

```
# Bug: Infinite loop
def loop_example():
    i = 0
    while i < 5:
        print(i)

40  # Task 5 (Infinite Loop - Wrong Condition)
41  # Bug: Infinite loop (FIXED)
42  def loop_example():
43      i = 0
44      while i < 5:
45          print(i)
46          i += 1
47
48  loop_example()
49
```

Output:

```
PS C:\Users\chara\OneDrive\Desktop\Ai-Assisted Coding>
ssignment-7.5.py"
0
1
2
3
4
```

Justification:

The loop became infinite because the loop variable was never updated. By incrementing i inside the while loop, the condition eventually becomes false, allowing the loop to terminate correctly.

Task - 6:

Prompt:

```
# Bug: Wrong unpacking  
a, b = (1, 2, 3)
```

```
51  # Task 6 (Unpacking Error – Wrong Variables)  
52  # Bug: Wrong unpacking (FIXED)  
53  a, b, c = (1, 2, 3)  
54  print(a, b, c)  
55
```

Output:

```
PS C:\Users\chara\OneDrive\Desktop\Ai-Assisted Coding>  
ssignment-7.5.py"  
1 2 3
```

Justification:

Tuple unpacking requires the number of variables to match the number of values. By providing three variables for the three elements in the tuple, the unpacking succeeds without raising a ValueError.

Task - 7:

Prompt:

```
# Bug: Mixed indentation  
def func():  
    x = 5  
    y = 10  
    return x+y
```

```
58  #TASK 7 Alternative: using _ to ignore extra values  
59  x, y, _ = (1, 2, 3)  
60  print(x, y)
```

Output:

```
PS C:\Users\chara\OneDrive\Desktop\Ai-Assisted Coding>  
ssignment-7.5.py"  
1 2
```

Justification:

When a tuple has more values than needed, Python allows using `_` as a throwaway variable. This safely ignores extra elements, prevents unpacking errors, and keeps the code clean and readable.

Task - 8:

Prompt:

```
# Bug: Wrong import
import maths
print(maths.sqrt(16))
63  # Task 8 (Import Error - Wrong Module Usage)
64  # Bug: Wrong import (FIXED)
65  import math
66  print(math.sqrt(16))
```

Output:

```
PS C:\Users\chara\OneDrive\Desktop\Ai-Assisted Coding>
ssignment-7.5.py"
4.0
```

Justification:

The error occurred because `maths` is not a valid Python standard library module. Importing the correct `math` module provides access to `sqrt()`, ensuring the program runs successfully and returns the correct result.